Parallel Image Processing Programming(ECE 7082)

Parallel programming using OpenMP-1

Xuenan Cui, xncui@inha.ac.kr

Computer Vision Lab.

School of Information & Communication Engineering

Inha University

Contents

- ❖ Introduction
- ❖ Directive and Clause
- ❖ Environment variables and Runtime library routines

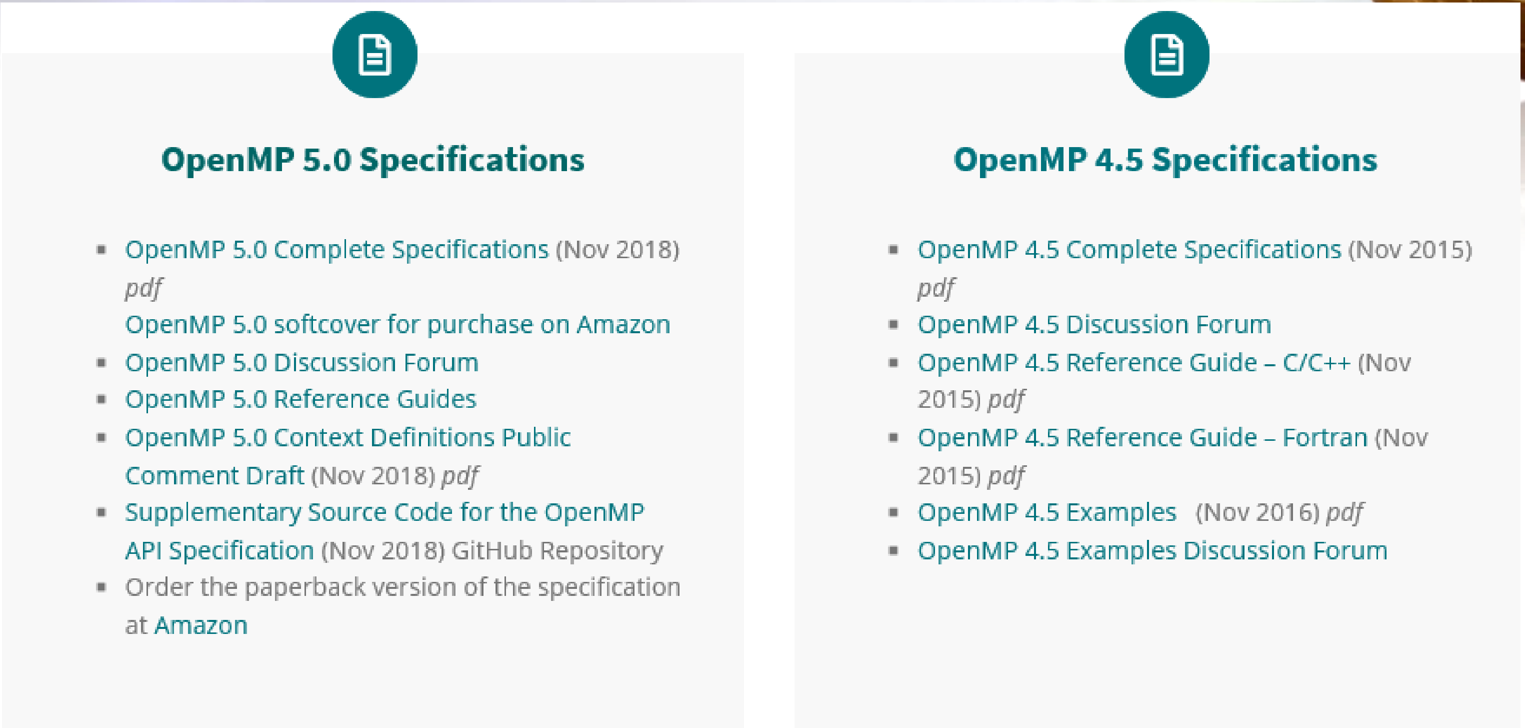
What is OpenMP

- Open Multi-Processing
- A standard developed under the review of many major software and hardware developers, government, and academia
  - Provide a standard among a variety of shared memory architectures/platforms
- OpenMP is not a new computer language; rather, it works in conjunction with either standard Fortran or C/C++

What is OpenMP

- OpenMP API is comprised of:
  - Compiler directives
  - Runtime library routines
  - Environment variables
- OpenMP support:
  - Fortran, C, C++
- Compilers supporting OpenMP:
  - Intel Compilers, Portland Group (PGI), IBM, Compaq
  - Omni, OdinMP can be used with gcc

OpenMP specifications

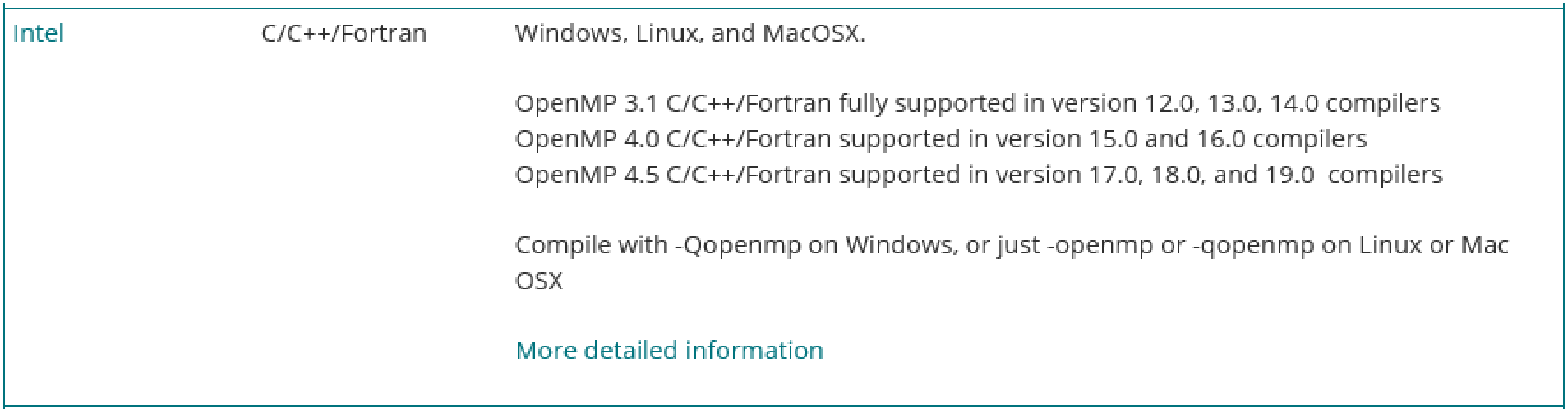


OpenMP 5.0 Specifications

- OpenMP 5.0 Complete Specifications (Nov 2018) *pdf*
  OpenMP 5.0 softcover for purchase on Amazon
- OpenMP 5.0 Discussion Forum
- OpenMP 5.0 Reference Guides
- OpenMP 5.0 Context Definitions Public Comment Draft (Nov 2018) *pdf*
- Supplementary Source Code for the OpenMP API Specification (Nov 2018) GitHub Repository
- Order the paperback version of the specification at Amazon

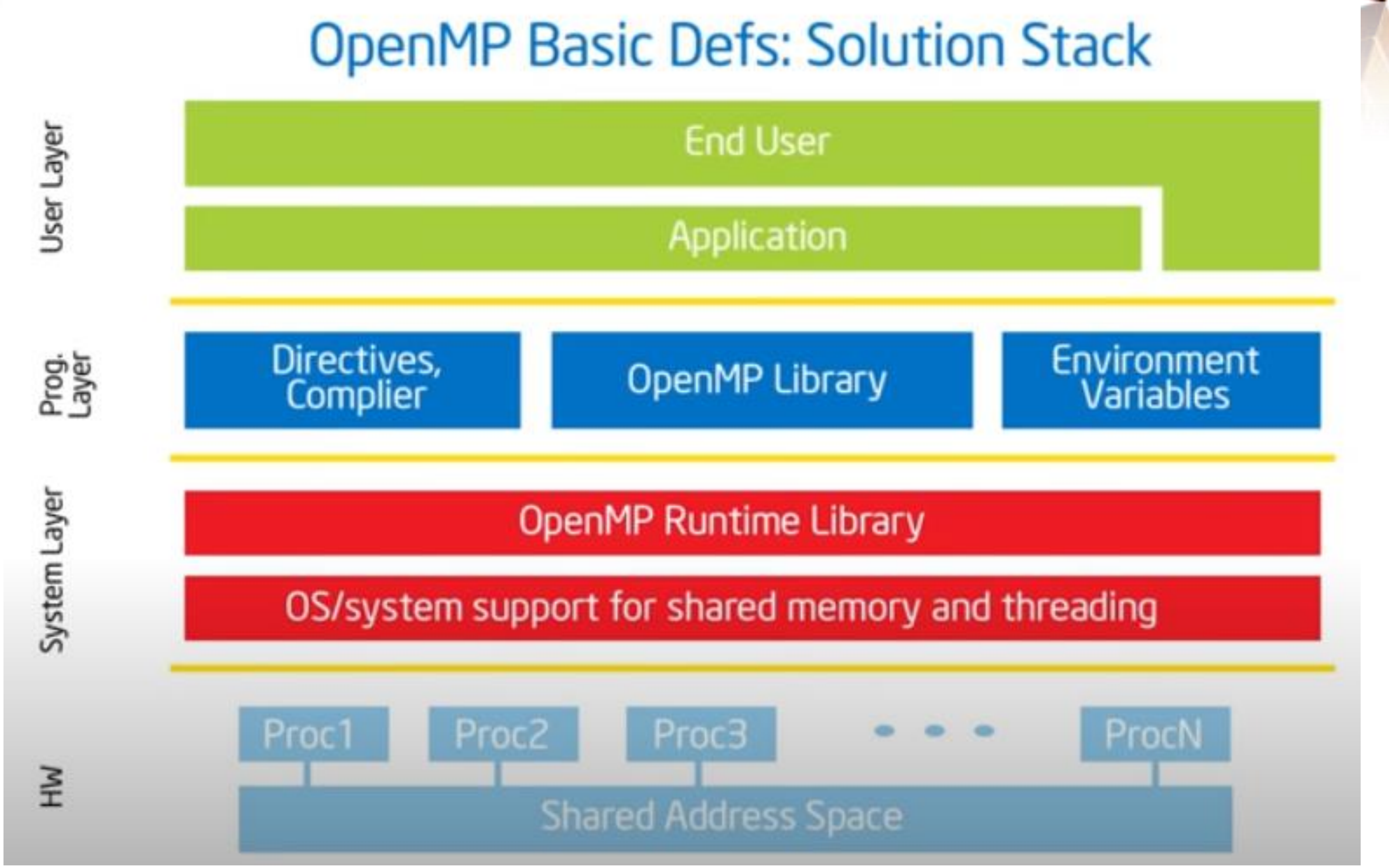
OpenMP 4.5 Specifications

- OpenMP 4.5 Complete Specifications (Nov 2015) *pdf*
- OpenMP 4.5 Discussion Forum
- OpenMP 4.5 Reference Guide – C/C++ (Nov 2015) *pdf*
- OpenMP 4.5 Reference Guide – Fortran (Nov 2015) *pdf*
- OpenMP 4.5 Examples (Nov 2016) *pdf*
- OpenMP 4.5 Examples Discussion Forum



Intel	C/C++/Fortran	Windows, Linux, and MacOSX. OpenMP 3.1 C/C++/Fortran fully supported in version 12.0, 13.0, 14.0 compilers OpenMP 4.0 C/C++/Fortran supported in version 15.0 and 16.0 compilers OpenMP 4.5 C/C++/Fortran supported in version 17.0, 18.0, and 19.0 compilers Compile with -Qopenmp on Windows, or just -openmp or -qopenmp on Linux or Mac OSX More detailed information

Solution Stack



OpenMP Basic Defs: Solution Stack

User Layer

End User

Application

Prog. Layer

Directives, Complier

OpenMP Library

Environment Variables

System Layer

OpenMP Runtime Library

OS/system support for shared memory and threading

HW

Proc1

Proc2

Proc3

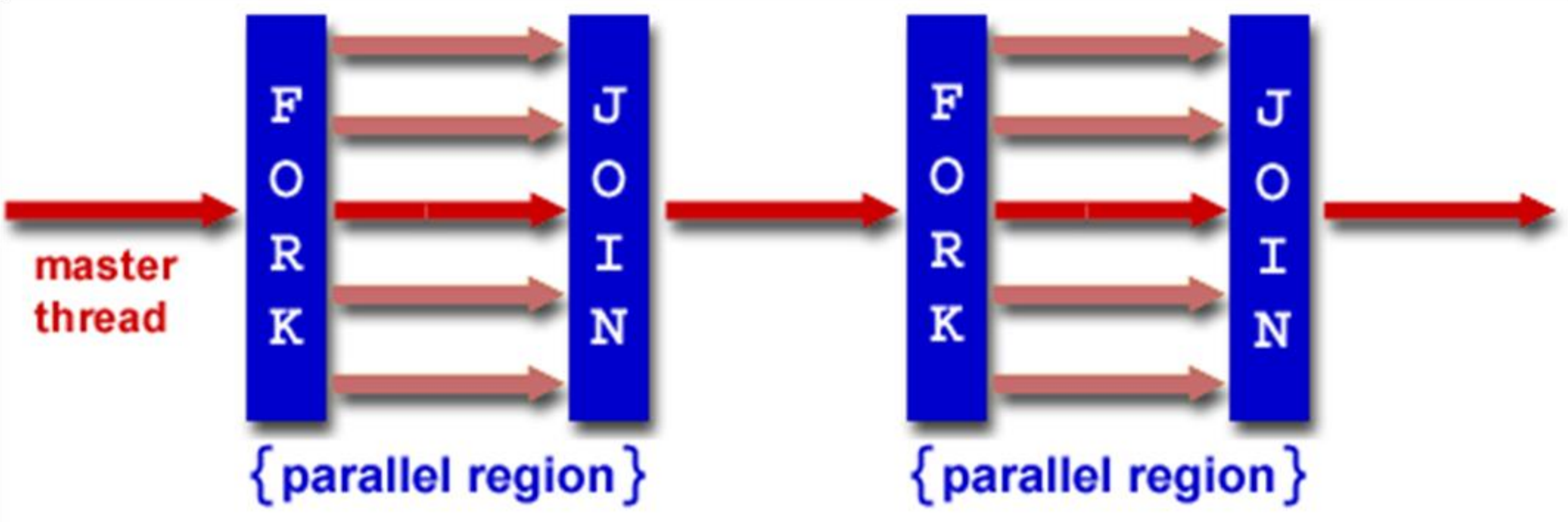
• • •

ProcN

Shared Address Space

OpenMP Programming Model

- Fork – Join Parallelism

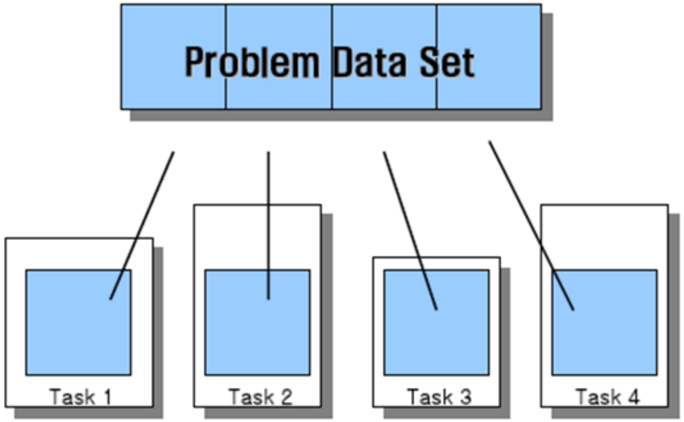


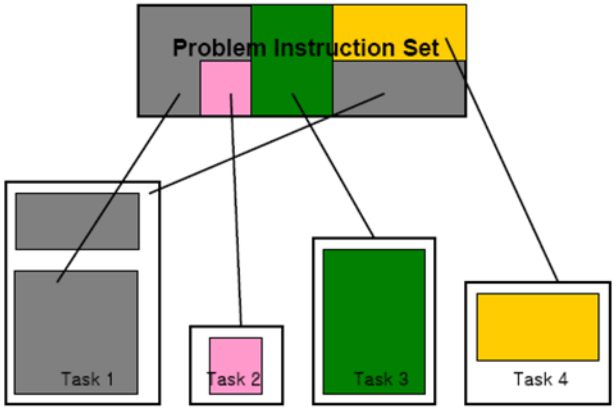
FORK: the master thread then creates a ***team* of parallel threads**

JOIN: When the team threads complete the statements in the parallel region construct, they synchronize and terminate, leaving only the master thread

Parallel Control Structures

- OpenMP provides two kinds of constructs for controlling, parallelism
  - Provides a directive to create multiple threads of execution that execute concurrently with each other
  - Provides constructs to divide work among an existing set of parallel threads

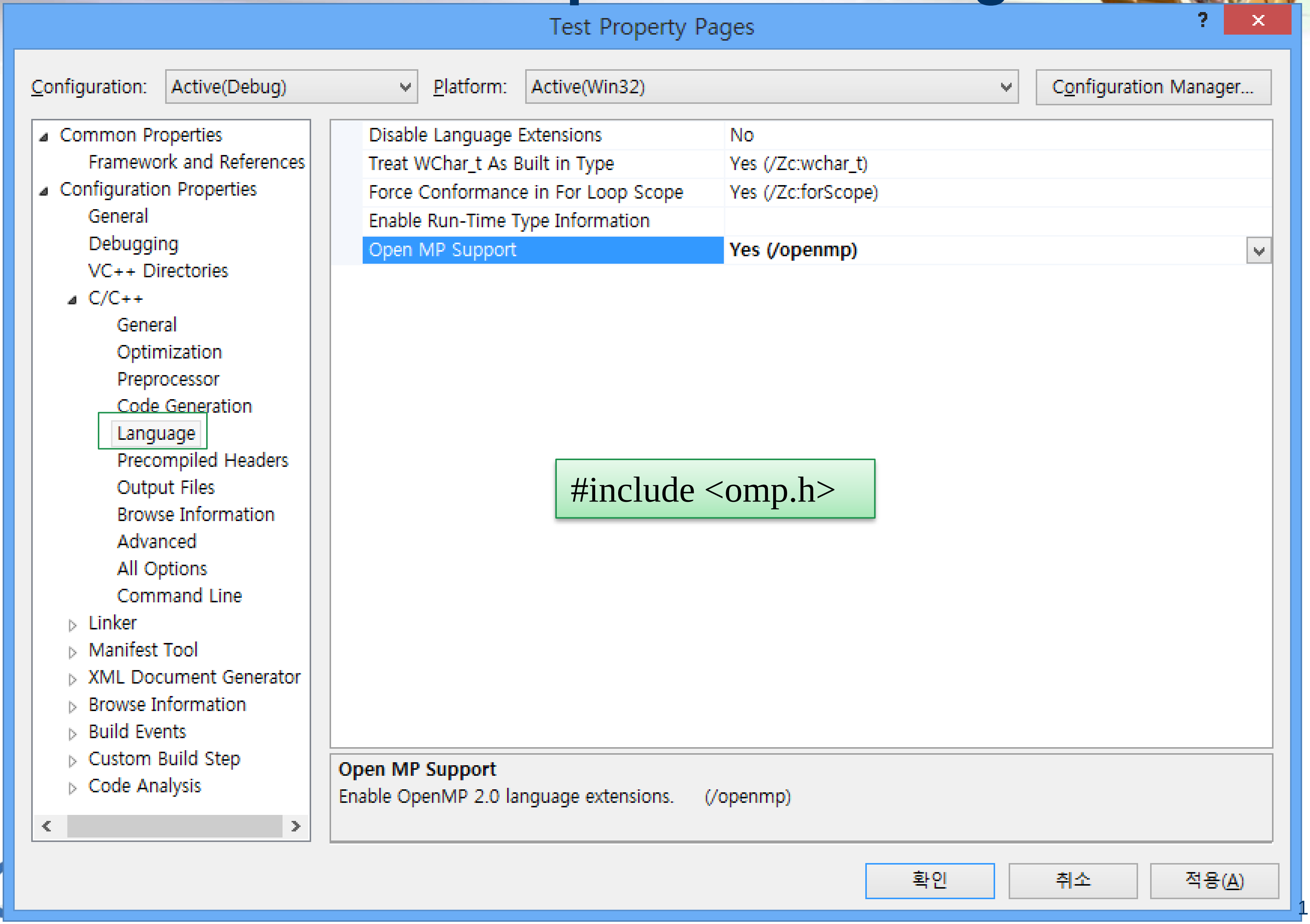




How Many Threads?

- The number of threads in a parallel region is determined by the following factors, in order of precedence:
  - Evaluation of the **IF** clause
  - Setting of the **NUM_THREADS** clause
  - Use of the **omp_set_num_threads()** library function
  - Setting of the **OMP_NUM_THREADS** environment variable
  - Implementation default - usually the number of CPUs on a node, though it could be dynamic
- Threads are numbered from 0 (master thread) to N-1

OpenMP Setting



Configuration: Active(Debug)

Platform: Active(Win32)

Disable Language Extensions	No
Treat WChar_t As Built in Type	Yes (/Zc:wchar_t)
Force Conformance in For Loop Scope	Yes (/Zc:forScope)
Enable Run-Time Type Information	
Open MP Support	**Yes (/openmp)**

Open MP Support

Enable OpenMP 2.0 language extensions.   (/openmp)

```
#include <omp.h>
```

Directive and Clause

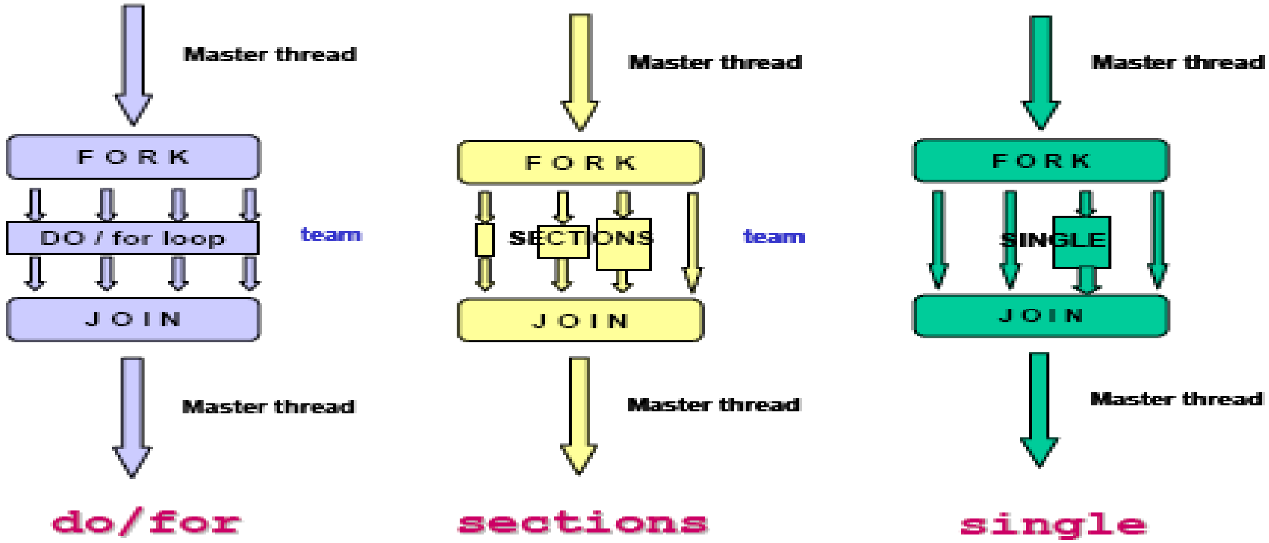
Directives Format

#pragma omp	directive-name	[clause, ...]	newline
Required for all OpenMP C/C++ directives.	A valid OpenMP directive. Must appear after the pragma and before any clauses.	Optional. Clauses can be in any order, and repeated as necessary unless otherwise restricted.	Required. Precedes the structured block which is enclosed by this directive.

#pragma omp parallel default(shared) private(beta, pi)

Working-Sharing

- *parallel/end parallel*
- OpenMP provides work-sharing directives
  - do/for, sections, single
  - Implied barrier in the End of the parallel part → synchronization

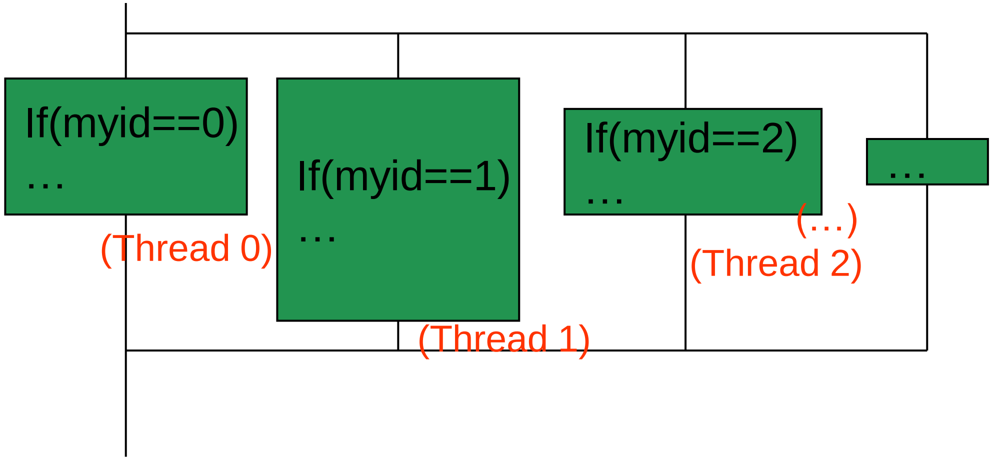


Working-Sharing (cont.)

- Form and usage of the *parallel* directive
  - It consist of a *parallel/end parallel* directive pair that can be used to enclose an arbitrary block of code
  - This directive pair specifies that the enclosed block of code, referred to as a parallel region, be executed in parallel by multiple thread

Working-Sharing (cont.)

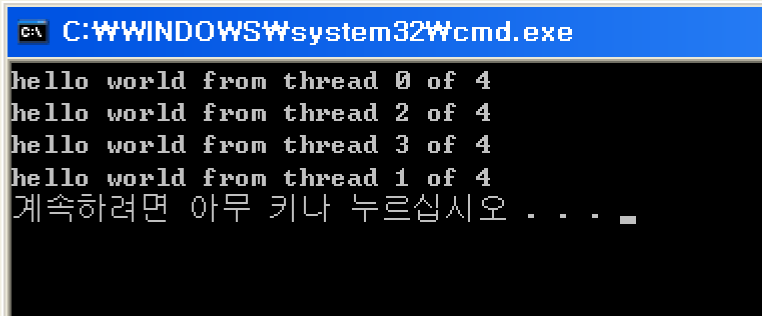
```
#pragma omp parallel
{
      myid=omp_get_thread_num();
      if(myid==0)
              do_something();
      else
              do_something(myid);
}
```



Divide the task using parallel directive

Work-Sharing Constructs

```
#pragma omp parallel
{
    printf( “hello world from thread %d of %d\n”,
    omp_get_thread_num(), omp_get_num_threads() );
}
```



```
hello world from thread 0 of 4
hello world from thread 2 of 4
hello world from thread 3 of 4
hello world from thread 1 of 4
계속하려면 아무 키나 누르십시오 . . . _
```

Work-Sharing Constructs (cont.)

```
double xyz[5000][3];
printf( “entering parallel region\n” );
#pragma omp parallel
{
        int tid;
        tid = omp_get_thread_num();
        compute_edges( tid, xyz );
}
printf( “parallel computation completed\n” );
```

Master Only

Thread Forks

Thread Private Space

Implicit barrier, Thread Join

Master Only

Note: xyz is shared between all threads!

Work-Sharing Constructs (cont.)

- #pragma omp for
  - Each thread receives a portion of work to accomplish - data parallelism
- #pragma omp section
  - Each section executed by a different thread-functional parallelism
  - Noniterative work-sharing
- #pragma omp single
  - Serialize a section of code, only one thread executes code block (good for I/O)

Do/for Directive

- Directive specifies that the iterations of the loop immediately following it must be executed in parallel by the team
- This assumes a parallel region has already been initiated, otherwise it executes in serial on a single processor.
- If you want do not wait other work, you can use nowait clause
  - threads do not synchronize at the end of the parallel loop

Do/for Directive (cont.)

```
#include <omp.h>
#define n 1000
main(){
  int i;
  double a[n], b[n], c[n];
  for(i=0; i<n; i++){
  a[i] = i*1.0;
  b[i] = a[i];}
#pragma omp parallel
{
#pragma omp for
{
for(i=0; i<n; i++){
c[i] = a[i] + b[i];}
}}}
```

Parallel code using *for* directive

Sections Directive

- OpenMP provides the *sections* work-sharing construct, which allows us to perform the entire sequence of tasks in parallel, assigning each task to a different thread
- The code for the entire sequence of tasks, or sections, begins with a sections directive and ends with an end sections directives
- In the *parallel* region, the beginning of each section is marked by *section* directive

Sections Directive (cont.)

```
#pragma omp parallel sections
{
#pragma omp section
   block
#pragma omp section
   block
#pragma omp section
   block
}
```

```
#include <omp.h>
Void main(){
#pragma omp parallel sections
{
#pragma omp section
printf( "hello world from thread %d of %d\n",
 omp_get_thread_num(),
omp_get_num_threads() );
#pragma omp section
printf( "hello world from thread %d of %d\n",
 omp_get_thread_num(),
omp_get_num_threads() );
}}
```

Parallel code using *sections* directive

Single Directive

- OpenMP provides the *single* construct to identify these kinds of tasks that must be executed by just one thread
- *Single* directive execute by first arrived thread
- Usually For data input/output

```
int len ;
double in[MAXLEN], out[MAXLEN], scratch[MAXLEN];
…
#pragma omp parallel shared(in, out, len)
{...
#pragma omp single
read_array(in, len);
#pragma omp for private(scratch)
for(j=1; j<=len; j++){
compute_result(out[j], &in, len, &scratch);}
#pragma omp single nowait
write_array(&out, len);}
```

Combined Directives

- *parallel for* directive
- *parallel sections* directive

Parallel + work-sharing	Combined
#pragma omp parallel #pragma omp for	#pragma omp parallel for
#pragma omp parallel #pragma omp sections	#pragma omp parallel sections

Clause

- *private(var1, var2, …)*
- *shared(var1, var2, …)*
- *default(shared|private|none)*
- *firstprivate(var1, var2, …)*
- *lastprivate(var1, var2, …)*
- *reduction(operator|intrinsic:var1, var2,…)*
- *schedule(type [,chunk])*
- *if(logical expression)ordered*

Private

- The *private* clause declares variables in its list to be private to each thread.
- *private*(*var1*,*var2*,…)
- *Private* variables behave as follows
  - A new object of the same type is declared once for each thread in the team
  - All references to the original object are replaced with references to the new object
  - Variables declared *private* should be assumed to be uninitialized for each thread

Private (cont.)

```
#pragma omp parallel for private(temp)
for(int i=0; i<=n; i++) {
    temp = 2.0*a[i];
    a[i] = temp;
    b[i] = c[i]/temp;
}
```

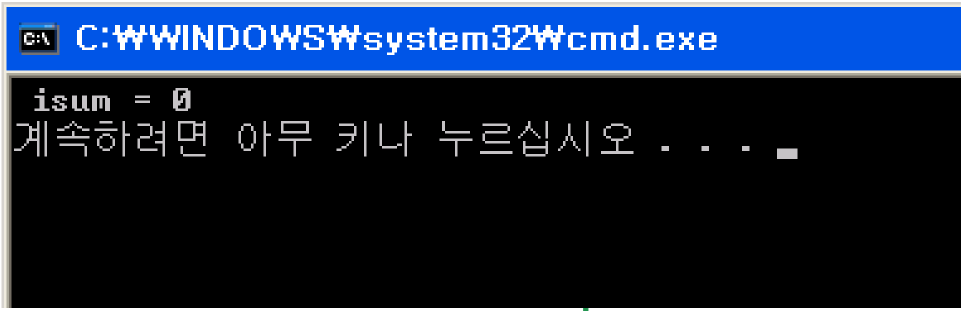
Shared,Default

- The *shared* clause declares variables in its list to be shared among all threads in the team
- *shared*(var1,var2,…)
- The *default* clause allows the user to specify a default scope for all variables in the lexical extent of any parallel region.
  - *default* (private|shared|none)

Firstprivate

- The *firstprivate* clause combines the behavior of the *private* clause with automatic initialization of the variables in its list.
- *firstprivate* (list)

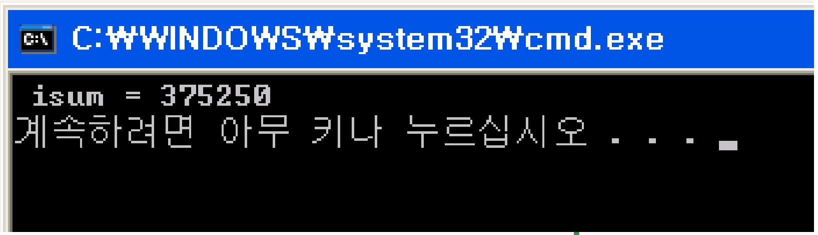
```
/* firstprivate */
main(){
int isum, i;
isum = 0;
#pragma omp parallel for firstprivate(isum)
for(i=1; i<=1000; i++)
/* 각 스레드는 각자 0 으로 초기화된 isum 값을 가진다 . */
isum = isum + i;
/* 그러나 , isum 은 여전히 정의되지 않는다 . */
printf( “ isum = %d \n” , isum);
}
```



```
isum = 0
계속하려면 아무 키나 누르십시오 . . .
```

Lastprivate

- The *Lastprivate* clause combines the behavior of the *private* clause with a copy from the last loop iteration or section to the original variable object.
- *Lastprivate* (list)



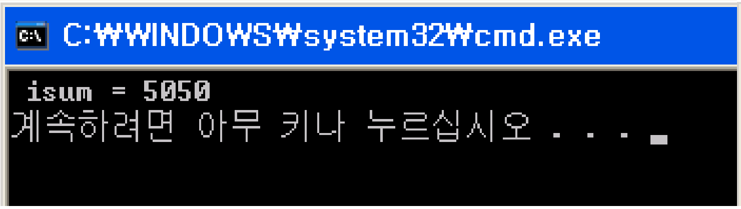
```
 isum = 375250
계속하려면 아무 키나 누르십시오 . . .
```

```
/* lastprivate */
main(){
int isum, i;
isum = 0;
#pragma omp parallel for firstprivate(isum) lastprivate(isum)
{
for(i=1; i<=1000; i++)
/* 각 스레드는 각자 0 으로 초기화된 isum 값을 가진다 . */
isum = isum + i;
}
/* 마지막 반복 (i=1000 일 때 ) 에서 계산되는 값을 출력한다 . */
printf( “ isum = %d \n” , isum);
}
```

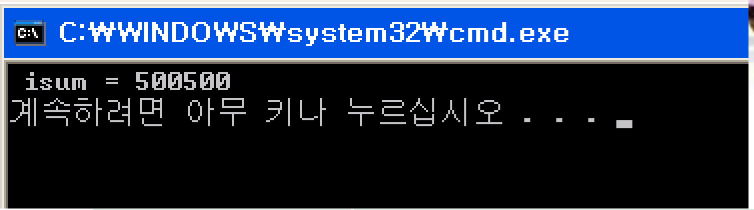
Reduction

- The *reduction* clause performs a reduction on the variables that appear in its list
- A private copy for each list variable is created for each thread.
- At the end of the reduction, the reduction variable is applied to all private copies of the shared variable, and the final result is written to the global shared variable
- ***reduction*** **(operator | intinsic:var1,var2,…)**

Reduction (cont.)

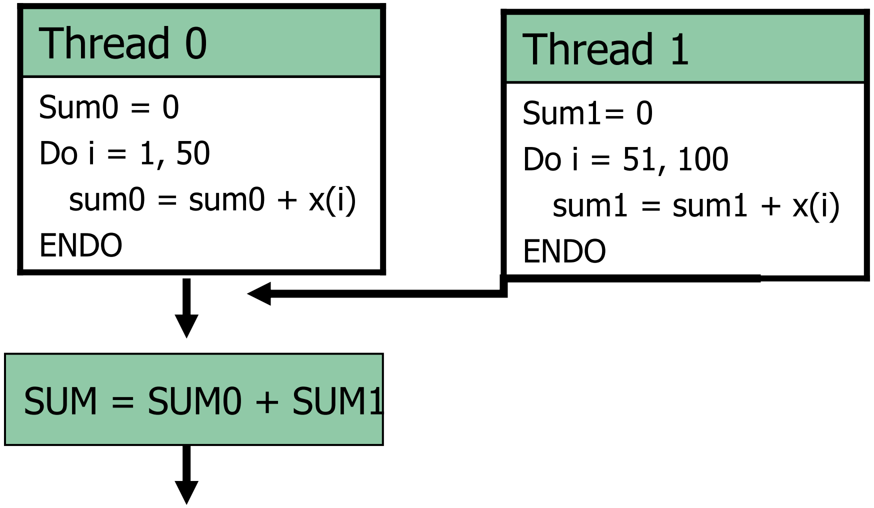


```
 isum = 5050
계속하려면 아무 키나 누르십시오 . . .
```



```
 isum = 500500
계속하려면 아무 키나 누르십시오 . . .
```

```
sum = 0;
#pragma omp parallel for reduction(+:sum)
for(i=1; i<=100; i++)
sum = sum + a[i];
```



```
Thread 0
Sum0 = 0
Do i = 1, 50
   sum0 = sum0 + x(i)
ENDO
```

```
Thread 1
Sum1= 0
Do i = 51, 100
   sum1 = sum1 + x(i)
ENDO
```

SUM = SUM0 + SUM1

Operator	Data Type	Initial Value
+	integer, floating point	0
*	integer, floating point	1
-	integer, floating point	0
&	integer	All bits on
|	integer	0
^	integer	0
&&	integer	1
||	integer	0

Schedule

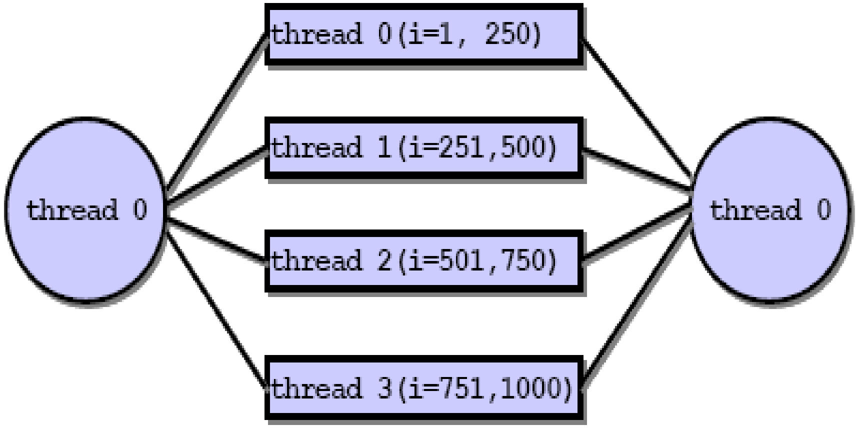
- Describes how iterations of the loop are divided among the threads in the team. The default schedule is implementation dependent
- schedule (type[,chunk])
  - static :
    - Loop iterations are divided into pieces of size *chunk* and then statically assigned to threads. If chunk is not specified, the iterations are evenly (if possible) divided contiguously among the threads

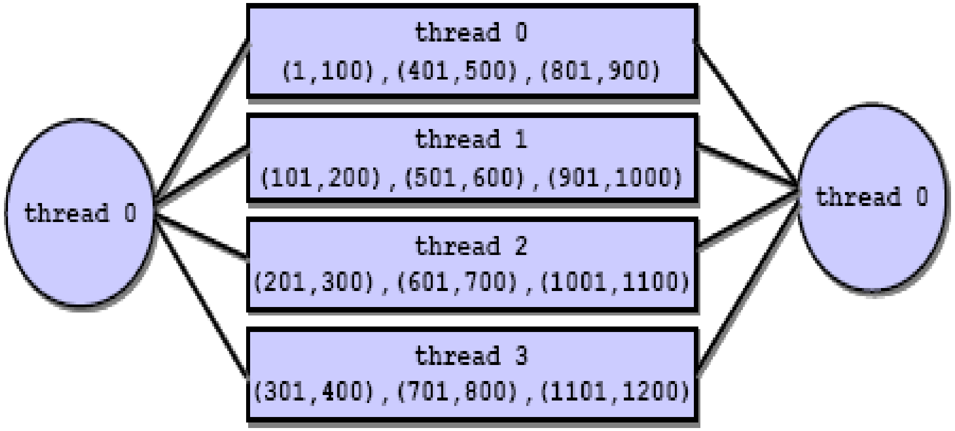
Schedule (cont.)

- Static schedule type

```
#pragma omp for shared(x) private(i) schedule(static)
      for(i=1;i<=1000;i++){
                  …work
}
```

```
#pragma omp for shared(x) private(i) schedule(static,100)
      for(i=1;i<=1200;i++){
                  …work
}
```





Schedule (cont.)

- schedule (type[,chunk])
  - dynamic :
    - Loop iterations are divided into pieces of size *chunk*, and dynamically scheduled among the threads;
    - When a thread finishes one chunk, it is dynamically assigned another. The default chunk size is 1

#prgma omp for shared(x) private(i) shcedule(dynamic, 1000)

Schedule(cont.)

- guided(Guided Self-scheduling:GSS):
  - First chunk size is defined value
  - Other chunk size is depend on rule
  - **size(chunk$_n$) = min(chunk_size, r^n * size(chunk$_0$))**
    - **Size(Chunk$_0$)= (# of iteration)/ (# of thread)**
    - **r = 1/#of thread**
  - The default chunk size is 1.
  - **#prgma omp for shared(x) private(i) shcedule(guided, 1000)**

Schedule(cont.)

- runtime:
  - The scheduling decision is deferred until runtime by the **environment variable** OMP_SCHEDULE
  - **export OMP_SCHEDULE= “static,1000”**
  - **export OMP_SCHEDULE= “dynamic”**

if

- When the condition is true, the program execute in parallel
- When the condition is false, the program execute in serial
- For parallel overhead

(a) if 문 사용

```
if(800 <= n){
        #pragma omp parallel for
        for(i=1, i<=n, i++)
        z[i] = a*x[i] + y;
}
else{
        for(i=1, i<=n, i++)
        z[i] = a*x[i] + y;
}
```

(b) if clause 사용

```
#pragma omp parallel for if (800 <= n)
for(i=1, i<=n, i++)
z[i] = a*x[i] + y;
```

Clauses with directives

Clause	Directives					
	Parallel	Do/for	Sections	Single	Parallel do/for	Parallel sections
IF	O				O	O
PRIVATE	O	O	O	O	O	O
SHARED	O	O			O	O
DEFAULT	O				O	O
FISRTPRIVATE	O	O	O	O	O	O
LASTPRIVATE		O	O		O	O
REDUCTION	O	O	O		O	O
COPYIN	O				O	O
SCHEDULE		O			O	
ORDERED		O			O	
NOWAIT		O	O	O		

Environment variables & Runtime library routines

Environment variables

- OpenMP provides the following environment variables for controlling the execution of parallel code

Environment variable	Description
OMP_DYNAMIC	Specifies whether the OpenMP run time can adjust the number of threads in a parallel region.
OMP_NESTED	Specifies whether nested parallelism is enabled, unless nested parallelism is enabled or disabled with **omp_set_nested**.
OMP_NUM_THREADS	Sets the maximum number of threads in the parallel region, unless overridden by *omp_set_num_threads* or *num_threads*.
OMP_SCHEDULE	Modifies the behavior of the *schedule* clause when **schedule(runtime)** is specified in a **for** or **parallel for** directive.

Environment variables

- Examples
  - OMP_SCHEDULE

    ```
    export OMP_SCHEDULE = “guided, 4”
    export OMP_SCHEDULE = “dynamic”
    ```

  - OMP_NUM_THREADS

    ```
    export  OMP_NUM_THREADS = 32
    ```

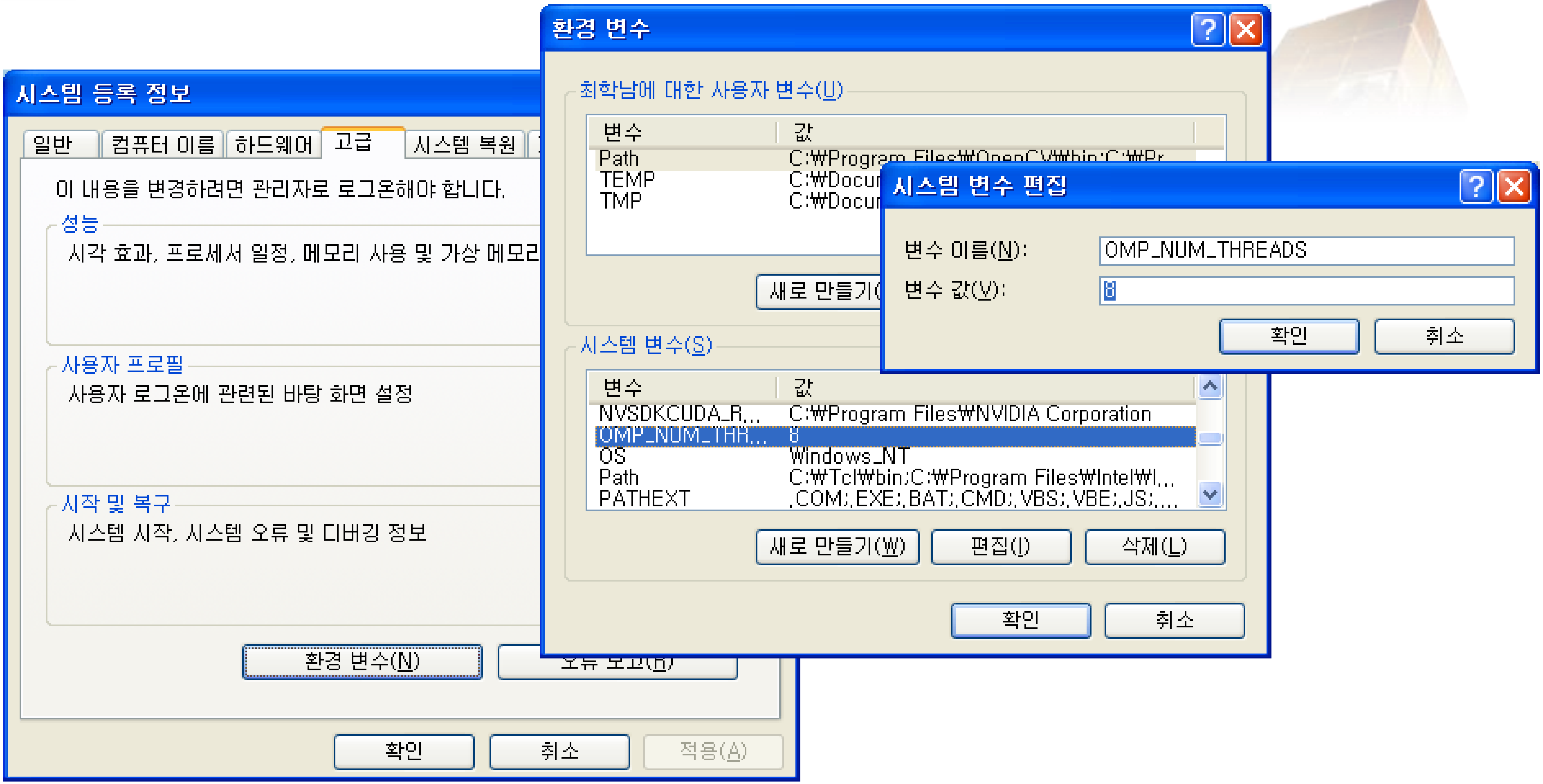
  - OMP_DYNAMIC

    ```
    export OMP_DYNAMIC = TRUE
    ```

  - OMP_NESTED

    ```
    export OMP_NESTED = FALSE
    ```

Environment variables



시스템 등록 정보

이 내용을 변경하려면 관리자로 로그온해야 합니다.

성능

시각 효과, 프로세서 일정, 메모리 사용 및 가상 메모리

사용자 프로필

사용자 로그온에 관련된 바탕 화면 설정

시작 및 복구

시스템 시작, 시스템 오류 및 디버깅 정보

환경 변수

최학남에 대한 사용자 변수(U)

변수	값
Path	C:₩Program Files₩OpenCV₩bin;C:₩Pr
TEMP	C:₩Docur
TMP	C:₩Docur

시스템 변수(S)

변수	값
NVSDKCUDA_R...	C:₩Program Files₩NVIDIA Corporation
OMP_NUM_THR...	8
OS	Windows_NT
Path	C:₩Tcl₩bin;C:₩Program Files₩Intel₩I...
PATHEXT	.COM;.EXE;.BAT;.CMD;.VBS;.VBE;.JS;...

시스템 변수 편집

변수 이름(N): OMP_NUM_THREADS

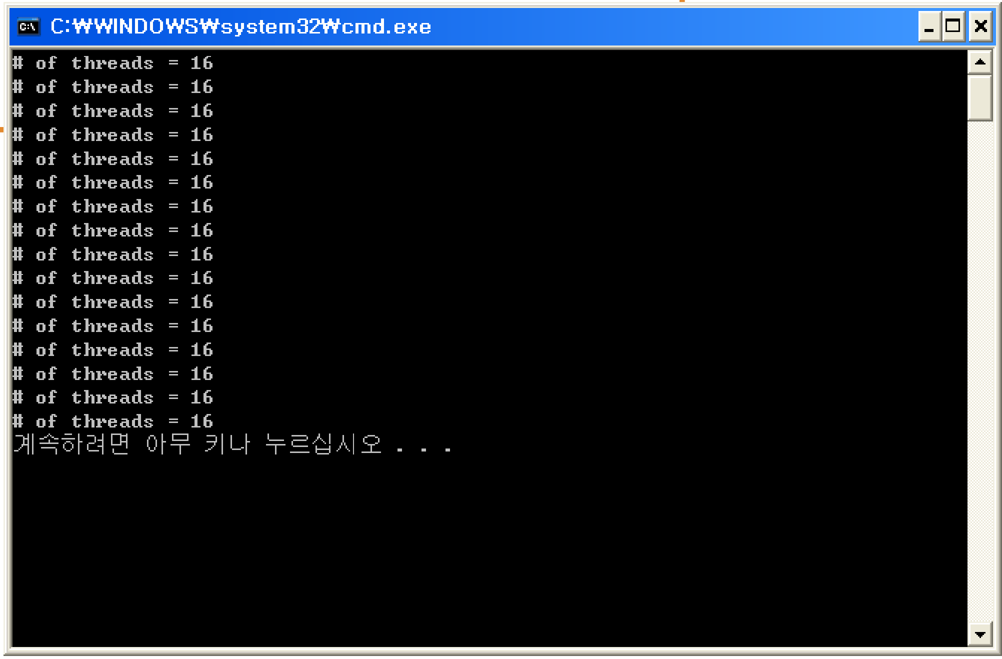
변수 값(V): 8

Runtime Library routines

- Execution environment routines
  - **void omp_set_num_threads(int *num_threads);***
    - Affects the number of threads used for subsequent **parallel** regions that do not specify a **num_threads clause.**
  - **int omp_get_num_threads(void);**
    - Returns the number of threads in the current team.
  - **int omp_get_max_threads(void);**
    - Returns maximum number of threads that could be used to form a new team using a “parallel” construct without a “num_threads” clause.

Runtime Library routines

```
#include<omp.h>
main(){
            num_threads = 16;
            omp_set_num_threads(num_threads);
            #pragma omp parallel
            {
            printf(“ # of thread=%d\n”, omp_get_num_threads() );
  }
 }
```



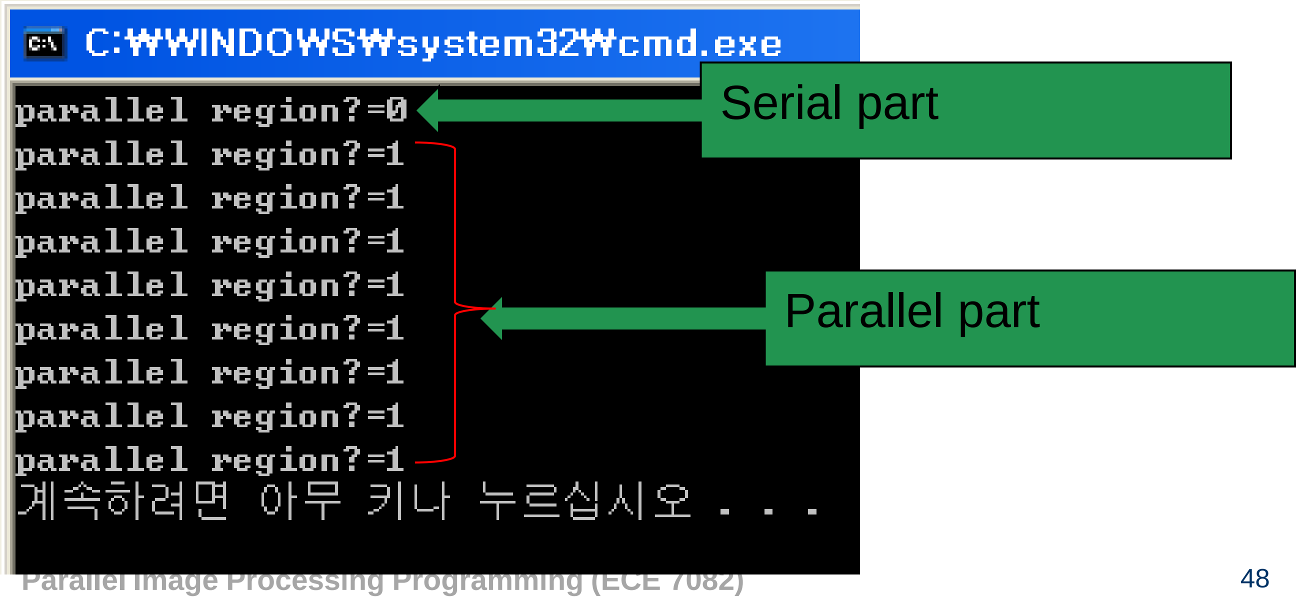
```
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
# of threads = 16
계속하려면 아무 키나 누르십시오 . . .
```

Runtime Library routines

- Execution environment routines
  - **int omp_get_thread_num(void);**
    - Returns the ID of the encountering thread where ID ranges from zero
    - to the size of the team minus 1.
  - **int omp_get_num_procs(void);**
    - Returns the number of processors available to the program.
  - **int omp_in_parallel(void);**
    - Returns *true if the call to the routine is enclosed by an active* **parallel region; otherwise, it returns *false*.**

Runtime Library routines

```
#include<omp.h>
  main(){
    printf("parallel region?=%d\n",omp_in_parallel());
     #pragma omp parallel
    {
                  printf("parallel region?=%d\n",omp_in_parallel());
     }
   }
```



```
parallel region?=0
parallel region?=1
parallel region?=1
parallel region?=1
parallel region?=1
parallel region?=1
parallel region?=1
parallel region?=1
parallel region?=1
계속하려면 아무 키나 누르십시오 . . .
```

Serial part

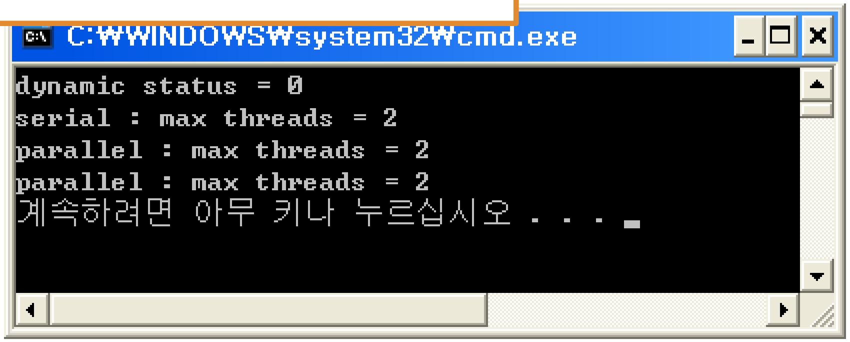
Parallel part

Runtime Library routines

- ❖ Execution environment routines
  - **void omp_set_dynamic(int *dynamic_threads);***
    - Enables or disables dynamic adjustment of the number of threads available.
    - omp_set_dynamic(1)
      - omp_get_num_threads() <= omp_get_max_threads()
    - omp_set_dynamic(0)
      - omp_get_num_threads() = omp_get_max_threads()
  - **int omp_get_dynamic(void);**
    - Returns the value of the *dyn-var internal control variable (ICV),* determining whether dynamic adjustment of the number of threads is enabled or disabled.
  - **void omp_set_nested(int *nested);***
    - Enables or disables nested parallelism, by setting the *nest-var ICV.*

Runtime Library routines

```
#include<omp.h>
  main(){
  printf(“dynamic status = %d\n”, omp_get_dynamic());
    printf(“serial : max threads = %d\n”,
             omp_get_max_threads());
  #pragma omp parallel
  {
    printf(“parallel : max threads = %d\m”,
                          omp_get_max_threads());
  }
 }
```



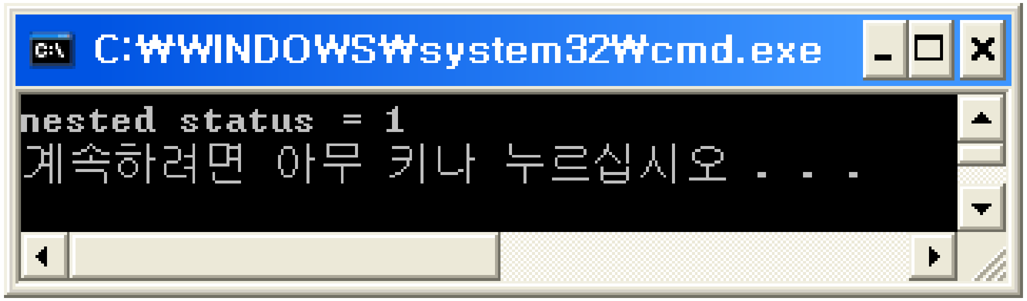
```
dynamic status = 0
serial : max threads = 2
parallel : max threads = 2
parallel : max threads = 2
계속하려면 아무 키나 누르십시오 . . . _
```

Runtime Library routines

- Execution environment routines
  - **int omp_get_nested(void);**
    - Returns the value of the *nest-var ICV, which determines if nested* parallelism is enabled or disabled.
  - **void omp_set_schedule(omp_sched_t *kind, int modifier);***
    - Affects the schedule that is applied when **runtime is used as**schedule kind, by setting the value of the *run-sched-var ICV.*
  - **void omp_get_schedule(omp_sched_t *kind,int *modifier);**
    - Returns the schedule applied when **runtime schedule is used.**

Runtime Library routines

```
#include<omp.h>
   main(){
    omp_set_nested(1);
    printf("nested status = %d\n",omp_get_nested());
   }
```



```
nested status = 1
계속하려면 아무 키나 누르십시오 . . .
```

Exercise

- Using parallel for to improve the following equations
- Using parallel sections to execute both equation at the same time
- Compare the processing between serial and parallel

  - S1=$\sum_{i=0}^{10000} i \ \ (int\ type)$
  - S2=$\prod_{i=1}^{20} i$ (*double type*)